Exhibit: Lightweight Structured Data Publishing

David F. Huynh, David R. Karger, Robert C. Miller
MIT Computer Science and Artificial Intelligence Laboratory,
The Stata Center, Building 32, 32 Vassar Street, Cambridge, MA 02139, USA
{dfhuynh, karger, rcm}@csail.mit.edu

ABSTRACT

It is no surprise that Semantic Web researchers and enthusiasts are excited to publish and accumulate semi-structured data on the Web. Beyond our community, however, we see many authors with structured data who want to publish it in rich browsing interfaces. These small-time authors are similar to early enthusiasts of the Web, simply excited by the opportunity to use a new medium to share information that they care about. For these users, we propose Exhibit, a lightweight structured data publishing framework that duplicates many of the desirable properties that contributed to the original growth of the Web. We argue that appealing to this segment of the Web population—addressing their publishing needs at very low cost—lets us leverage their labor to put structure on content that otherwise would be published in hand-authored HTML, and thus very hard to harvest automatically.

Categories and Subject Descriptors

H5.2 [Information interfaces and presentation]:
User Interfaces – Graphical user interfaces (GUI).
H5.4 [Information interfaces and presentation]:
Hypertext/Hypermedia – User issues.

General Terms

Design, Human Factors.

Keywords

Web, publish, filter, sort, faceted browsing, dynamic query, HTML, DOM, generated UI, lens, view, template.

1. INTRODUCTION

This paper describes *Exhibit*, a very lightweight AJAX framework that lets relatively unskilled individuals create web pages containing rich, dynamic visualizations of structured data and supporting faceted browsing and sorting on that structured data. With passing knowledge of HTML, people can use Exhibit to quickly build structured data-centric web pages and unintentionally contribute to the Semantic Web without having to install, configure, and maintain any database, and without having to write a single line of server-side code. They only have to be passionate about some structured data that they wish to publish and Exhibit will make publishing that structured data almost as easy as publishing unstructured documents.

1.1 Motivation

A search in Google for "breakfast cereals" turns up as the first hit not a commercial or corporate site, but rather, Topher's Breakfast Cereal Character Guide [9], a homemade site run by a single person. Topher's site has won media honors and recognitions since 1997, and has a high page rank as a result. But Topher's site, like web sites of many early adopters of the Web, still looks and behaves like it was made in 1997, rather than in 2006. While most commercial and institutional sites are now database-backed, serving up sophisticated browsing, searching, and visualizing user interfaces, homemade sites still consist of small sets of static HTML pages, lacking advanced features that web users have now come to expect.

Consider yet other authors:

- A professor wants to let visitors to his web site sort his 97 publications by year or group them by research projects, conferences, or co-authors.
- A history high-school teacher wants to showcase 57 important discoveries of bronze age tools to her students through a web site that renders both maps and time lines.
- A group of politically conscious citizens have observed unsettling patterns in the coming election and would like to report their evidence in illuminating ways for the whole world to see.

For these authors, putting up RDF/XML data files will not meet their goals because few users have tools able to access that data. Furthermore, the authors may be as interested in the precise *presentation* of their data as they are in the data itself—they want to give users the right pathways for structured browsing and rich visualizations. But even to support a basic feature like sorting, they need to create a database, design a schema, design the user interface, write server-side code, write client-side code, and make sure that their three-tier web *application*—overflowing with technical jargon like SQL, PHP, CSS, and straddling between the web server and many different browsers and platforms—works as expected.

On the early Web, it was easy for enthusiastic but relatively unskilled individuals to stake out territories on the Web by copying other web pages and inserting their own data. These *homesteaders* were crucial contributors to the early growth of the Web. Their homesteads slowly grew in size and complexity, with each homesteader incrementally acquiring specific additional skills needed to meet their growing ambitions for rich presentation. In contrast, today's hopeful homesteaders must pick up a whole alphabet soup of tools before they can begin publishing structured content. It is no wonder that structured browsing and rich presentations are offered mostly by commercial sites and large institutions who can afford web site engineering costs, or by the few technically-savvy who have the whole package of web technologies under their belts.

1.2 Approach

We argue that with the right tools, people who want to publish structured data—whether or not they care about the Semantic Web—will be motivated and able to adopt the rich presentations offered by our tools, and will thus slip into becoming homesteaders of the Semantic Web alongside Semantic Web enthusiasts.

Our Exhibit framework duplicates several key features that enabled the growth of the original Web:

- **No installation, configuration, or maintenance.** Once a web server was up, anyone could "join the web" simply by putting an HTML file on the server. Similarly with Exhibit, an author joins the Semantic Web simply by putting an HTML file and a data file on their web server, and their site's visitors immediately experience that data through a rich browsing interface.
- **Copy and paste evolution.** Any author who wishes to publish their data can do so simply by copying someone else's Exhibit files and replacing specific elements with their own data. The replacement can be done by simple pattern matching, without any understanding of the formal syntax of the Exhibit system.
- **Incremental complexity.** As authors get more excited about the system, they can add additional functionality and complexity in small steps—they never need to swallow a whole new set of ideas in one dose. At the same time, there are few limits placed on what creative users can do with their material.
- **No network effect required.** Unlike social networking sites, Exhibit provides immediate benefits to its first adopter, regardless of others' actions. It simplifies the author's management of their information collection, and offers visitors using existing web browsers a better interface to that collection than can be built by typical web authoring tools with the same efforts.

Our contribution is two-fold: Exhibit (1) offers small-scale authors access to the same types of rich interactive browsing of structured content currently available only on large organizational web sites and (2) gives such authors significant incentives without significant barriers to adopt the technology, and as a side effect makes their content available in structured form, where it can be contributes to the growth of the Semantic Web.

2. RELATED WORK

From one point of view, Exhibit is designed to enliven home-made web pages with advanced features such as sorting and filtering, and richer visualizations including time lines and maps. Here, related work includes Chickenfoot [12] and Greasemonkey [5], which let users write scripts to modify web pages being displayed in their browsers to enhance the pages for their personal use; and Sifter [15], which automatically extracts items from a sequence of related pages and offers a faceted browsing interface on those items right within the original web page. While these efforts expect users to do the work needed to augment web pages, Exhibit targets authors. Compared to users, authors have more expertise about their own data and care more about organizing and presenting it well.

Web widgets such as Google Maps [4] and Timeline [7] let authors offer rich visualizations and interactions on their sites. These widgets have been used far and wide, but only by individuals who run their own servers and understand the Javascript programming necessary to integrate the widgets with their data. Exhibit makes use of these widgets itself, but lets non-programming non-administrative content authors embed them by adding a few tags to their HTML documents, without touching the server..

There have also been numerous pieces of research work on faceted browsing [13, 16, 17, 18, 20], many from the Semantic Web community. While Exhibit provides a faceted browsing interface, that is not our contribution in this paper. The interface simply serves to illustrate user interface riches made possible without server reconfiguration when the structured data within a web page is separated from its presentational elements.

On the authoring structured data front, traditional spreadsheet software satisfy this need of non-technical users. Software for making small databases, such as Microsoft Access, might be user-friendly enough for some people. However, the data created by these applications cannot be exported out and published as a dynamic web page or web site.

Wikis let authors input structured data through custom templates, and the Semantic MediaWiki extension [19] extends MediaWiki's wikitext syntax to allow authors to enter arbitrary Semantic Web annotations into wiki pages. These tools require installation, setup, and maintenance—start-up costs that Exhibit does not have.

DabbleDB [2], Google Base [3], and the likes let people enter or upload structured data into online databases owned and managed by the respective companies. The flexibility in personalizing the presentation of that data on these web services cannot match the level of personalization achievable by coding HTML. Furthermore, not every author likes to publish her own data through another company's web site.

Existing applications for generating web sites that support structured browsing, e.g., online photo albums, are all domain-specific. Exhibit is domain-independent. These applications work by processing data supplied by the author and churning out whole web sites containing several web pages hard-wired together. The author can select one of several themes, but there is little further personalization of the output that can be done compared to Exhibit.

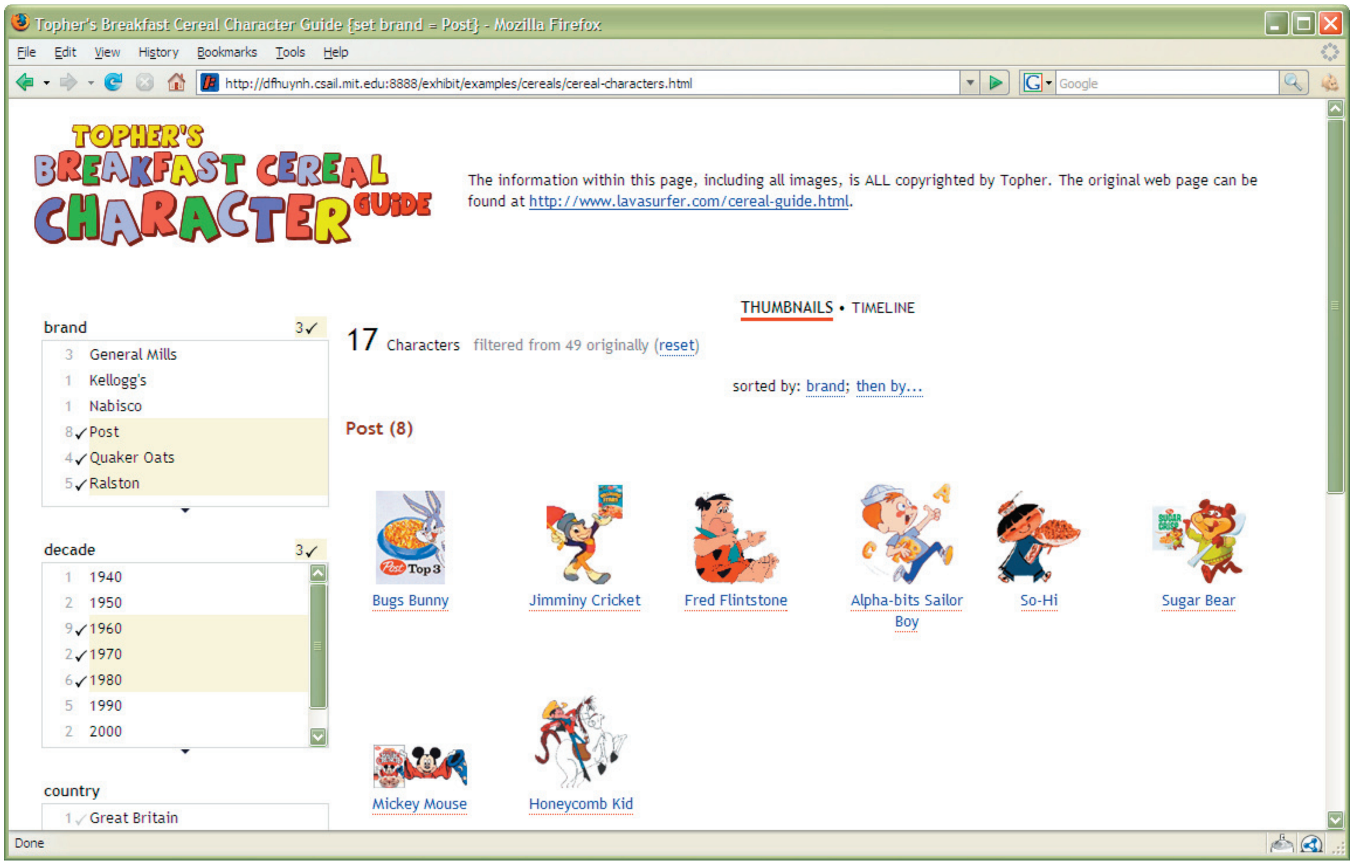
3. INTERFACE DESIGN

As Exhibit is a publishing framework, it has two interfaces: one facing every user of the published information and one facing the author. A web page that embeds Exhibit will be referred to as an *exhibit* in lowercase hereafter.

3.1 User Interface

An exhibit looks just like any other web page, except that it has more advanced features mostly seen on commercial and institutional sites. Figure 1 and Figure 2 show two exhibits covering different types of information. Each is styled differently, but there are several common elements, as described below.

Exhibit's user interface consists of three panels: the browse panel, the view panel, and the control panel, whose locations on the page are controlled by the author. The view panel displays a collection of items in one or more switchable *views*. The exhibit in Figure 1 is configured by the author to support two different views of the same data: THUMBNAILS and TIMELINE. THUMBNAILS is currently selected by the user and it is showing. The exhibit in Figure 2 is configured to support six views, and the BIRTH PLACES view is currently showing. Each kind of view—map, timeline, table, thumbnail, and tile—supports its own configuration settings.


Topher's Breakfast Cereal Character Guide {set brand = Post} - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://dfhuynh.csail.mit.edu:8888/exhibit/examples/cereals/cereal-characters.html
Google
TOPHER'S BREAKFAST CEREAL CHARACTER GUIDE
THUMBNAILS • TIMELINE
Done

The information within this page, including all images, is ALL copyrighted by Topher. The original web page can be found at http://www.lavasurfer.com/cereal-guide.html.

17 Characters filtered from 49 originally (reset)

sorted by: brand; then by...

brand	3✓
3	General Mills
1	Kellogg's
1	Nabisco
8 ✓	Post
4 ✓	Quaker Oats
5 ✓	Ralston

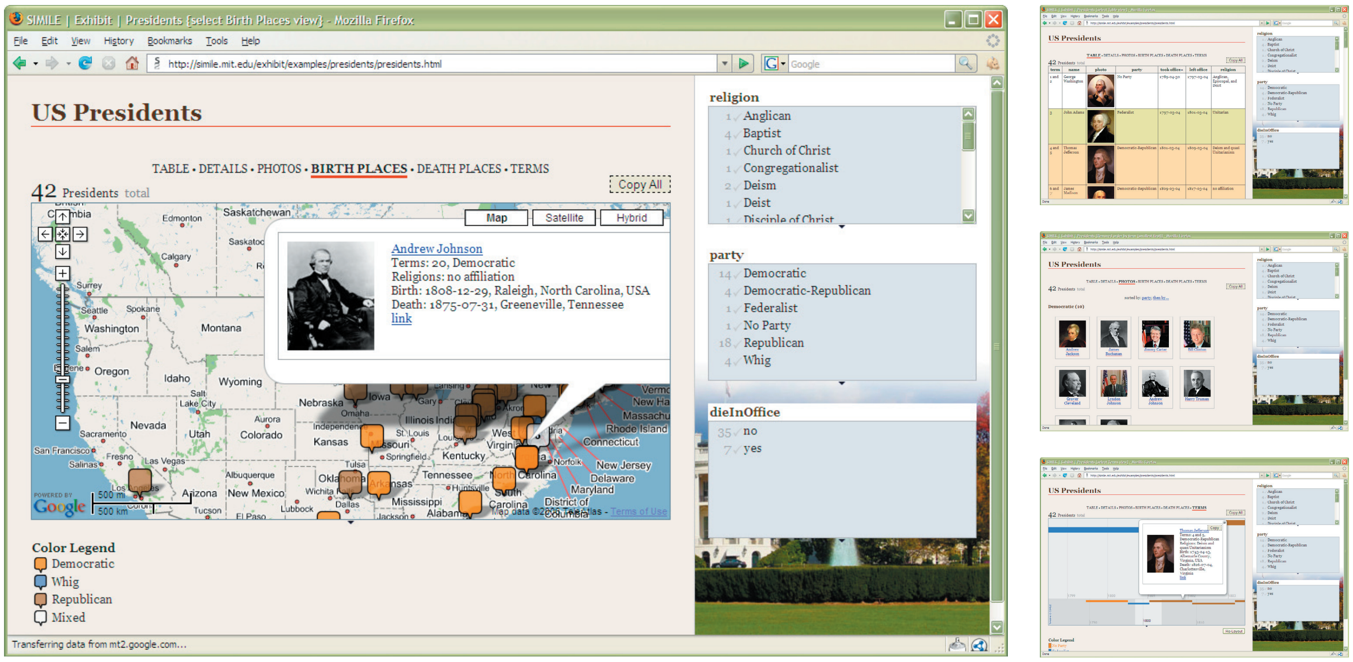
decade	3✓
1	1940
2	1950
9 ✓	1960
2 ✓	1970
6 ✓	1980
5	1990
2	2000

country	
1	Great Britain

Post (8)

Bugs Bunny · Jimminy Cricket · Fred Flintstone · Alpha-bits Sailor Boy · So-Hi · Sugar Bear · Mickey Mouse · Honeycomb Kid

Figure 1. A web page embedding Exhibit to show information about breakfast cereal characters. The information can be viewed as thumbnails or on a timeline and filtered through a faceted browsing interface. Topher, the original author of the information, has eagerly agreed to host this exhibit on his own web site.


SIMILE | Exhibit | Presidents {select Birth Places view} - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://simile.mit.edu/exhibit/examples/presidents/presidents.html
Google
US Presidents
TABLE • DETAILS • PHOTOS • BIRTH PLACES • DEATH PLACES • TERMS
Copy All
Map Satellite Hybrid
Transferring data from mt2.google.com...

42 Presidents total

Andrew Johnson
Terms: 20, Democratic
Religions: no affiliation
Birth: 1808-12-29, Raleigh, North Carolina, USA
Death: 1875-07-31, Greeneville, Tennessee
link

Color Legend
- Democratic
- Whig
- Republican
- Mixed

religion	
1	Anglican
4	Baptist
1	Church of Christ
1	Congregationalist
2	Deism
1	Deist
	Disciple of Christ

party	
14	Democratic
4	Democratic-Republican
1	Federalist
1	No Party
18	Republican
4	Whig

dieInOffice	
35	no
7	yes

Figure 2. A web page embedding Exhibit to show information about U.S. presidents in 6 ways, including maps, table, thumbnails, and timelines.

The Google Maps [4] map in the BIRTH PLACES view is configured to color-code its markers by the political parties of the presidents being mapped. Although these settings are specified by the author, some can be changed dynamically by the user (e.g., sorting order in Figure 1). Exhibit's user interface can be extended by third-parties' views if the author chooses to included them.

Items can be presented differently in different views. Where there is little space to render sufficient details in-place (e.g., on a map), markers or links provide affordance for popping up bubbles containing each item's details (e.g., map bubble in Figure 2). The rendition of each item contains a link for bookmarking it individually. Invoking this link later will load the exhibit and pop up a rendition of the bookmarked item automatically.

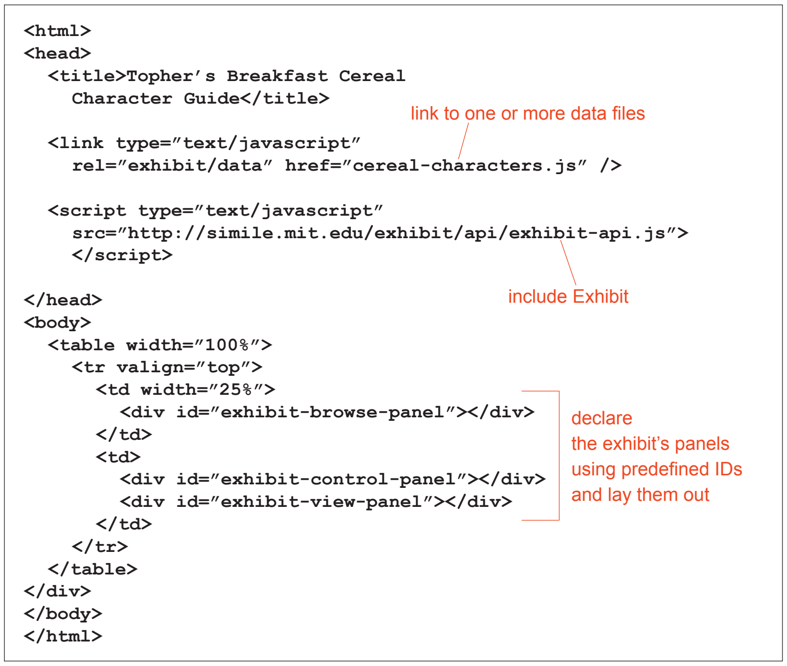
The browse panel (left in Figure 1 and right in Figure 2) contains facets by which users can filter the items in the view panel. This is a conventional dynamic query interface with preview counts. The control panel is currently unused. In the future, it will host UI controls that let the user manipulate the exhibit's data and presentation in more generic ways than the author has intended.

The reader is encouraged to visit http://simile.mit.edu/exhibit/ to try out several live exhibits, including:

- Topher's Breakfast Cereal Character Guide
- US Presidents
- Dinosaurs
- MIT Nobel Prize Winners
- The Kennedy Family

```
{
  items: [
    { type:      ‘Character’,
      label:     ‘Trix Rabbit’,
      brand:     ‘General Mills’,
      decade:    1960,
      country:   [ ‘USA’, ‘Canada’ ],
      thumbnail: ‘images/trix-rabbit-thumb.png’,
      image:     ‘images/trix-rabbit.png’,
      text:      ‘First appearing on ...’
    },
    // ... more characters ...
  ]
}
```

Figure 3. An Exhibit JSON data file showing data for one breakfast cereal character, which is encoded as property/value pairs.



```
<html>
<head>
  <title>Topher’s Breakfast Cereal
    Character Guide</title>

  <link type=”text/javascript”
    rel=”exhibit/data” href=”cereal-characters.js” />

  <script type=”text/javascript”
    src=”http://simile.mit.edu/exhibit/api/exhibit-api.js”>
    </script>

</head>
<body>
  <table width=”100%”>
    <tr valign=”top”>
      <td width=”25%”>
        <div id=”exhibit-browse-panel”></div>
      </td>
      <td>
        <div id=”exhibit-control-panel”></div>
        <div id=”exhibit-view-panel”></div>
      </td>
    </tr>
  </table>
</div>
</body>
</html>
```

Figure 4. To create the web page in Figure 1, the author starts with this boiler plate HTML code, which displays the characters in `cereal-characters.js` through the default lens that lists property/value pairs.

3.2 Author Interface

Making an exhibit like Figure 1 involves two tasks: authoring the data and authoring the presentation. Both are iterated until the desired result is achieved. This section briefly describes the publishing process, leaving technical details to later sections.

3.2.1 Creating the Data

Exhibit currently can read data in its own JSON [6] format. The data file for those breakfast cereal characters looks something like Figure 3. The items' data is coded as an array of objects containing property/value pairs. Values can be strings, numbers, or booleans. If a value is an array, then the corresponding item is considered to have multiple values for that property. For instance, according to Figure 3, the Trix Rabbit character is released in both the U.S. and in Canada.

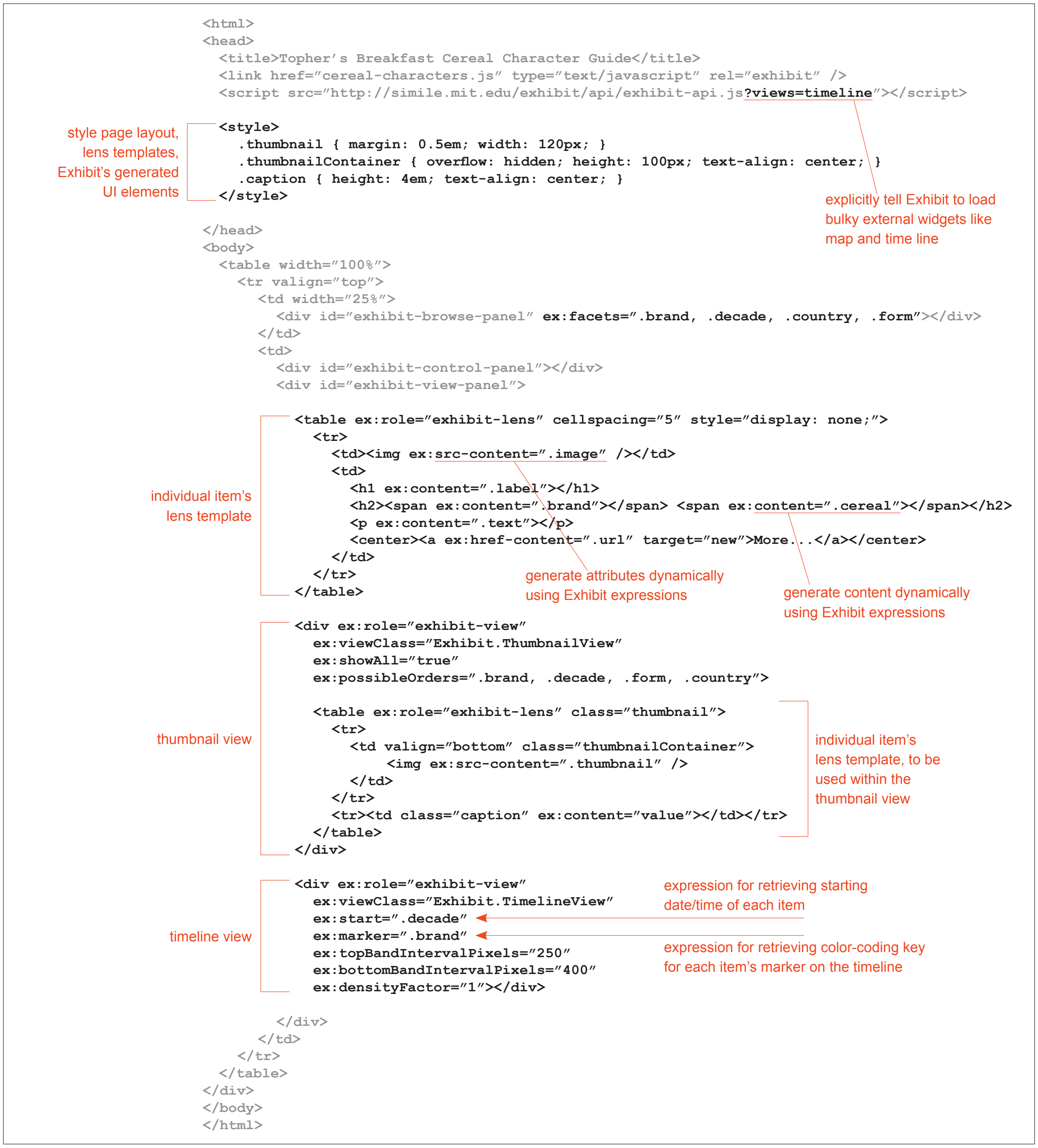
The author is mostly free to make up the names of the properties. We will discuss the data model in the next major section.

Data for a single exhibit need not reside in a single file. It can be split into multiple files for convenience. For example, a couple's recipes exhibit can store their data in two files: her-recipes.json and his-recipes.json. The exhibit just needs to load both.

Exhibit data files can be edited in any text editor. We also offer a web service called Babel [1] through which authors can convert various formats to the Exhibit JSON format and even preview the results in a generic exhibit. Babel can currently convert between RDF/XML, N3, Bibtex, Tab Separated Values, and Exhibit JSON. The reader is encouraged to upload his or her own data to Babel, or cut and paste rows of tab-separated values from a spreadsheet into Babel, to experiment with Exhibit.

3.2.2 Creating the Presentation

The web page itself is just a regular HTML file that can be created locally, iterated locally until satisfaction, and then, if desired, uploaded together with the data files to the web server. Figure 4 shows the initial HTML code needed to start making the exhibit in Figure 1. This code instantiates an Exhibit instance, loads it with the data file referenced by the first **`<link>`** element, and configures the exhibit's view panel to show a tile view. The tile view, by default, sorts all items in the exhibit by labels and displays the top ten items using the default *lens*. This lens shows property/value pairs for each item. A lot of reasonable defaults are hardwired into Exhibit to give the author some result with minimal initial work.



```
<html>
<head>
  <title>Topher's Breakfast Cereal Character Guide</title>
  <link href="cereal-characters.js" type="text/javascript" rel="exhibit" />
  <script src="http://simile.mit.edu/exhibit/api/exhibit-api.js?views=timeline"></script>

  <style>
    .thumbnail { margin: 0.5em; width: 120px; }
    .thumbnailContainer { overflow: hidden; height: 100px; text-align: center; }
    .caption { height: 4em; text-align: center; }
  </style>

</head>
<body>
  <table width="100%">
    <tr valign="top">
      <td width="25%">
        <div id="exhibit-browse-panel" ex:facets=".brand, .decade, .country, .form"></div>
      </td>
      <td>
        <div id="exhibit-control-panel"></div>
        <div id="exhibit-view-panel">

          <table ex:role="exhibit-lens" cellspacing="5" style="display: none;">
            <tr>
              <td><img ex:src-content=".image" /></td>
              <td>
                <h1 ex:content=".label"></h1>
                <h2><span ex:content=".brand"></span> <span ex:content=".cereal"></span></h2>
                <p ex:content=".text"></p>
                <center><a ex:href-content=".url" target="new">More...</a></center>
              </td>
            </tr>
          </table>

          <div ex:role="exhibit-view"
            ex:viewClass="Exhibit.ThumbnailView"
            ex:showAll="true"
            ex:possibleOrders=".brand, .decade, .form, .country">

            <table ex:role="exhibit-lens" class="thumbnail">
              <tr>
                <td valign="bottom" class="thumbnailContainer">
                    <img ex:src-content=".thumbnail" />
                </td>
              </tr>
              <tr><td class="caption" ex:content="value"></td></tr>
            </table>
          </div>

          <div ex:role="exhibit-view"
            ex:viewClass="Exhibit.TimelineView"
            ex:start=".decade"
            ex:marker=".brand"
            ex:topBandIntervalPixels="250"
            ex:bottomBandIntervalPixels="400"
            ex:densityFactor="1"></div>

        </div>
      </td>
    </tr>
  </table>
</div>
</body>
</html>
```

explicitly tell Exhibit to load bulky external widgets like map and time line
style page layout, lens templates, Exhibit's generated UI elements
individual item's lens template
generate attributes dynamically using Exhibit expressions
generate content dynamically using Exhibit expressions
thumbnail view
individual item's lens template, to be used within the thumbnail view
timeline view
expression for retrieving starting date/time of each item
expression for retrieving color-coding key for each item's marker on the timeline

Figure 5. The author starts with the `code in gray` (from Figure 4), includes more and more of the `code in black`, and tweaks until the desired result (Figure 1) is achieved (logo graphics and copyright omitted). Tweaking involves following online documentation or just copying code from other existing exhibits. When more expressivity is needed than this declarative syntax allows for, a bit of Javascript can be added (Figure 7).

The author does not even need to write this initial HTML code from scratch: it is trivial to copy this code from existing exhibits or from online tutorials. This is how HTML pages are often made—by copying existing pages, removing unwanted parts, and incrementally improving until satisfaction. The declarative syntax of HTML, the forgiving nature of web browsers and their reasonable defaults, and the quick HTML edit/test cycle make HTML authoring easy. We designed Exhibit to afford the same behavior.

Figure 5 shows the final HTML code needed to render the exhibit in Figure 1 (logo graphics and copyright message omitted). The additional code, in black, configures the facets in the browse panel, the two views in the view panel, one all-purpose lens, and one lens to be used in the THUMBNAILS view.

Making the presentation look better can also involve filling in and fixing up the schema. Figure 6 shows how the plural label for the type **`Character`** is declared so that plural labels in the UI, e.g., 12 Characters, can be generated properly. The **`decade`** property values are declared to be dates instead of strings so that they can be sorted as dates.

To change the schema, e.g., renaming a property, the author can simply invoke the text editor's Replace All command. Or if the data is in a spreadsheet, she can just rename the corresponding column header label. Saving old versions of the data involves making copies of the data files. Changing schema and versioning data might not be as simple if databases were used.

Should more expressivity be required than the declarative syntax allows for, a bit of Javascript code can be added to configure the exhibit. Figure 7 shows how to configure a table view that contains images and custom color-coded rows.

4. DATA MODEL

Although Exhibit data model *instances* are RDF graphs, but the *abstract* Exhibit data model is a *sub*-model of RDF (as detailed below). Hence, not all RDF data models can be expressed easily in Exhibit. This design choice allows for specialized, simple syntax (based on JSON) for rapid authoring. For simple, small data sets, we believe that Exhibit data models are sufficiently expressive, yet much easier to write than general-purpose RDF/XML or N3.

An Exhibit data model contains a set of *items*, each having a *type* and several *properties*.

4.1 Items

Items in each exhibit are identified by IDs unique within that exhibit. That is, IDs serve the same role as URIs, but in the closed world of one exhibit.

So that items within exhibits can exported out into the Web and still retain a globally unique identity, each item is assigned a URI either explicitly by the author (by adding a **`uri`** property value to the item) or automatically generated by appending its ID to the URL of the exhibit where it originates. Even though each item has a URI, it is still referenced locally by its ID because its ID should be much shorter and more human-readable than its URI.

In fact, the author does not even have to make up the ID for each item in her exhibit. If an item has no **`id`** property value, its ID is taken to be the same as its **`label`** property value. Note that in Figure 3, the item has neither **`id`** nor **`uri`** property value; both values will be generated.

Each item must have a **`label`** property value, which is a string naming that item in a human-friendly manner (subject to the author's discretion). This label is used to render the item wherever a concise, textual description of that item is needed.

```
{
  types: {
    'Character': {
      pluralLabel: 'Characters'
    }
  },
  properties: {
    'url': {
      valueType: "url"
    },
    'decade': {
      valueType: "date"
    }
  }
  items: [
    // ... items ...
  ]
}
```

Figure 6. Schema information can be added to the JSON file to improve Exhibit's user interface.

Hence, in most cases, the author is only burdened to make up labels that are readable to other humans rather than globally unique identifiers for the sake of machines; the former is less cognitively demanding than the latter. We consider this departure from RDF to be an improvement in this context of lightweight structured data publishing.

4.2 Types

Each item has a type, which can be specified by the author as the item's **`type`** property value, or if missing, defaults to the generic type **`Item`** (analogous to **`owl:Thing`**). Just like items, types are also identified locally by IDs and globally by URIs; these IDs and URIs can also be explicitly declared or generated in the same manner as items' IDs and URIs.

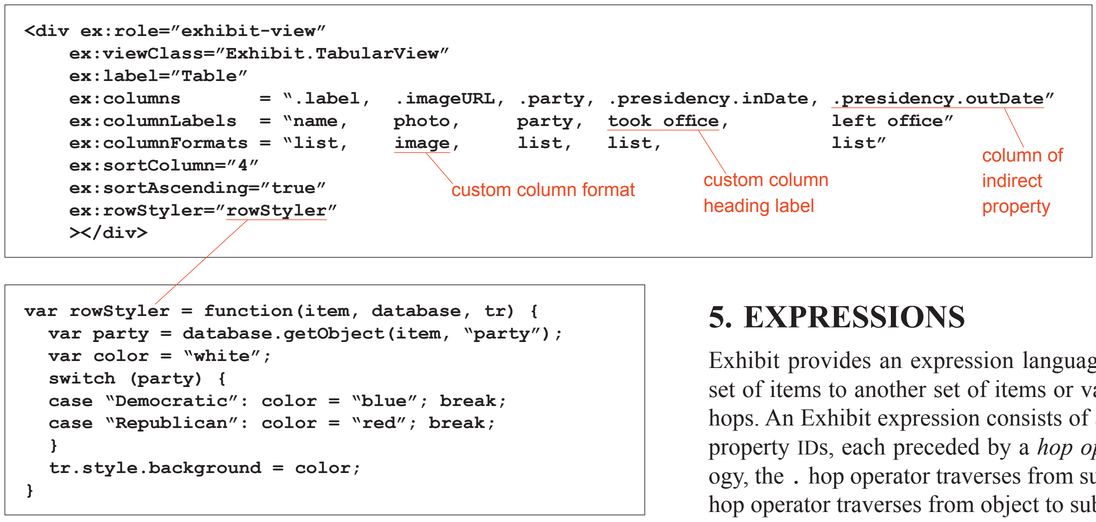
Beside **`id`**, **`uri`**, and **`label`**, a type has one more property, **`pluralLabel`**, which is used to generate more grammatical user interface text (e.g., 9 People instead of 9 Person). If **`pluralLabel`** is not explicitly declared, its value is the same as the **`label`** property value.

In fact, there is no need for the author to explicitly declare every type in the schema. A type is added to the system whenever an item of that type is added. This lets the author focus on the items—the main point of her exhibit—and only add types and properties when they make the user interface better.

4.3 Properties

Properties are also given human-readable labels and identified locally by IDs and globally by URIs in the same manner as items and types. In addition to **`id`**, **`uri`**, and **`label`**, a property also has other fields: **`reverseLabel`**, **`pluralLabel`**, **`reversePluralLabel`**, **`groupingLabel`**, and **`reverseGroupingLabel`**. These additional labels are used to generate more user-friendly UI text, e.g., child of rather than reverse of parent of.

Property values—the equivalence of objects in RDF statements—are not typed individually. That is, the values of each property must all be text strings, or must all be numbers, or dates, or booleans, etc.; or they must all be item IDs. In RDF terminology,



```
<div ex:role="exhibit-view"
    ex:viewClass="Exhibit.TabularView"
    ex:label="Table"
    ex:columns       = ".label,  .imageURL, .party, .presidency.inDate, .presidency.outDate"
    ex:columnLabels  = "name,    photo,     party,  took office,        left office"
    ex:columnFormats = "list,    image,     list,   list,               list"
    ex:sortColumn="4"
    ex:sortAscending="true"
    ex:rowStyler="rowStyler"
    ></div>
```

```
var rowStyler = function(item, database, tr) {
  var party = database.getObject(item, "party");
  var color = "white";
  switch (party) {
  case "Democratic": color = "blue"; break;
  case "Republican": color = "red"; break;
  }
  tr.style.background = color;
}
```

Figure 7. Advanced configuration of a tabular view. Mixing HTML and Javascript yields the best of both worlds: familiarity, simplicity, and forgiving nature of HTML's declarative syntax to start, and expressiveness of Javascript's imperative syntax whenever needed.

all values of a property must be literals of the same XSD type, or they are all resources (with potentially different `rdf:type`). This departure from RDF compromises data modeling flexibility for editing convenience: there is one single place to change the types of all values of a property. In most cases, and especially in small, simple data sets, this compromise is sensible.

There is a fixed set of value types, currently including: `text`, `number`, `date`, `boolean`, `url`, and `item`. Figure 6 shows how property value types are declared in data files. All values of type `item` are locally unique IDs of items in the same exhibit. When an Exhibit data model is converted into RDF, values of types other than `item` are converted to RDF literals, while values of type `item` are converted to resources by interpreting those values as item IDs and retrieving the `URI` property values of the corresponding items.

This design choice of tying value types to properties, rather than requiring type declaration on each value, facilitates incremental improvements to the data. For instance, `brand` property values in Figure 3 are initially, by default, of type `text`. This might satisfy the author until she wants to record details about the brands, at which time she can simply specify the value type for `brand` to be `item` and adds the data for the brands (Figure 8).

4.4 Data Management

Our choice for a data model simpler than RDF makes it easier for authors to manage their data. Managing data in a text file is acceptable for a few dozen items, but perhaps not for a few hundred items. However, data sets of such sizes can be managed in spreadsheets, copied off or saved in Tab Separated Value format, and converted into the Exhibit JSON format using Babel [1]. If an item has several values for one particular property, the values are separated by semicolons in the one single cell.

5. EXPRESSIONS

Exhibit provides an expression language for traversing from one set of items to another set of items or values through one or more hops. An Exhibit expression consists of a sequence of one or more property IDs, each preceded by a *hop operator*. In RDF terminology, the `.` hop operator traverses from subject to object while the `!` hop operator traverses from object to subject. For example,

- evaluating `.hasAuthor.teachesAt.locatedIn` on some papers returns the locations of the schools where the authors of those papers teach;
- evaluating `.spouseOf!parentOf` on some people returns their parents-in-laws.
- evaluating `!shot!arrested` on John F. Kennedy returns the police officers who arrested his assassinator.

Figure 7 shows a few expressions being used to specify columns of a tabular view.

Currently, this expression language can only traverse the data model, and in that way, it resembles the Fresnel Selector Language [11]. In the future, we plan to extend the language to support computations so that, for example, the author can specify

`.lastName + ", " + .firstName` and
`yearOf(now) - yearOf(.birthDate)`

as columns in a tabular view.

```
{
  properties: {
    'brand': {
      valueType: 'item'
    }
  },
  items: [
    { type:      'Character',
      label:     'Trix Rabbit',
      brand:     'General Mills',
      decade:    1960,
      country:   [ 'USA', 'Canada' ],
      thumbnail: 'images/trix-rabbit.png',
      text:      'First appearing on ...'
    },
    // ... more characters ...

    { type:        'Brand',
      label:       'General Mills',
      headQuarter: 'Minnesota',
      founded:     1856
    }
    // ... more brands ...
  ]
}
```

Figure 8. Elaboration of brand property values by specifying the value type for the property `brand` and adding Brand items.

This expression language is currently sufficient for configuring various parts of Exhibit. In the future when the needs arise, a SPARQL [8] interface can be added.

6. LENSES AND VIEWS

An Exhibit lens renders one single item while an Exhibit view renders a set of items, possibly by composing several lenses in some layout. Exhibit comes default with several views: tile view, thumbnail view, tabular view, time line view, and map view. Third parties can add more views. Figure 2 shows a few of these views. Each view has its own settings, which vary based on the complexity of the view. For example, the map view needs an expression that specifies how to retrieve the latitude/longitude pair for each item and an expression that retrieves some property which can be used to color code the map markers; the tile view, on the other hand, needs to know how to sort the items.

While views come pre-built (but are configurable), lenses can be written by coding lens *templates*. A template is just a fragment of HTML that can be specified in-line, as in Figure 9, or in a different file.

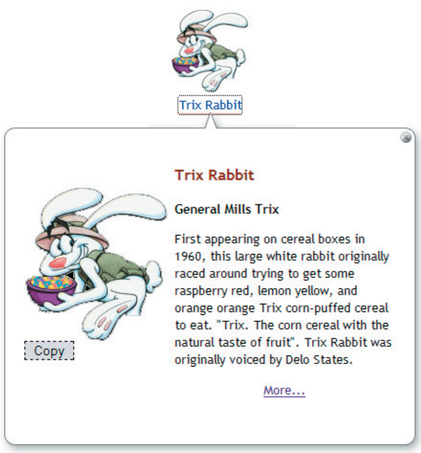
Within a lens template, the **content** attribute of an element specifies what content to stuff into that element when the template is instantiated for an item. For example, in Figure 9, the **<h1>** element will be filled with the **label** property value of the item.

Attributes that end with **-content** are assumed to contain Exhibit expressions. These expressions are resolved into actual values when the lens is instantiated, and the values are used to assert HTML attributes of the same name but without the **ex:** namespace and the **-content** suffix. For example, the **ex:src-content** attribute in Figure 9 is replaced with the **image** property value of the item being rendered, generating the attribute **src="images/trix-rabbit.png"** in the generated HTML **<img>** element.

The **if-exists** attribute of an element determines whether that element and its children in the template should be included in the presentation of a particular item. For example, if an item does not have an **image** property value, the template in Figure 9 will not generate a broken image.

The **control** attribute specifies which Exhibit-specific control to embed. There are only two controls supported at the moment: the **item-link** control and the **copy-button** control; the first is a permanent link to the item being rendered and the latter is a drop-down menu button that lets the user copies the item's data off in various formats.

Note that Exhibit adopts a *templating* approach to generating lenses. In contrast, Fresnel [11] takes a *styling* approach in which styling rules are declared separately and then applied to morph a generic presentation into a custom presentation.



Trix Rabbit

General Mills Trix

First appearing on cereal boxes in 1960, this large white rabbit originally raced around trying to get some raspberry red, lemon yellow, and orange orange Trix corn-puffed cereal to eat. "Trix. The corn cereal with the natural taste of fruit". Trix Rabbit was originally voiced by Delo States.

```
<table ex:role="exhibit-lens" cellspacing="5"
    style="display: none;">
  <tr>
    <td>
      <img ex:if-exists=".cereal"
          ex:src-content=".image" />
      <div ex:control="copy-button"></div>
    </td>
    <td>
      <h1 ex:content=".label"></h1>
      <h2>
        <span ex:content=".brand"></span>
        <span ex:content=".cereal"></span>
      </h2>
      <p ex:content=".text"></p>
      <center ex:if-exists=".url">
        <a ex:href-content=".url"
          target="new">More...</a>
      </center>
    </td>
  </tr>
</table>
```

Figure 9. Lens template for showing a breakfast cereal character in a pop-up bubble.

7. EXPORTERS

The Copy button in Figure 9 and the Copy All button in Figure 2 pop up menus that let the user pick a format in which to export one or more items. Exhibit currently provides exporters for RDF/XML, Exhibit JSON, Semantic MediaWiki extension wikitext [19], and Bibtex. The author can register third parties' exporters with Exhibit so that they show up in these menus.

The purpose of these exporters is to facilitate and encourage propagation of structured data by offering convenience to both users and authors. For example, being able to copy off the Bibtex of some publications that you have found in an exhibit so that you can cite them is very convenient. You can also copy that data in Exhibit JSON format and incorporate it into your own exhibit to make an archive of related work.

Exhibit's default exporters generate an **origin** property value for each item to export. This value is the URL that, when invoked, returns to the original exhibit and pops up the view of that item automatically. This is a lightweight mechanism for attribution.

8. IMPLEMENTATION

The Exhibit framework is implemented in several Javascript, CSS, and image files. It is available at a public URL where anyone can reference it from within his or her HTML pages. Exhibit authors do not need to download any software, and exhibit viewers do not need to install any browser extension. This zero cost is the signature of client-side include Web APIs and is largely responsible for the explosion in the embedding use of Google Maps [4].

Exhibit's source code is available publicly. Any of its parts can be overridden by writing more Javascript code and CSS definitions after including Exhibit's code. Third parties can implement additional views and exporters to supplement our library.

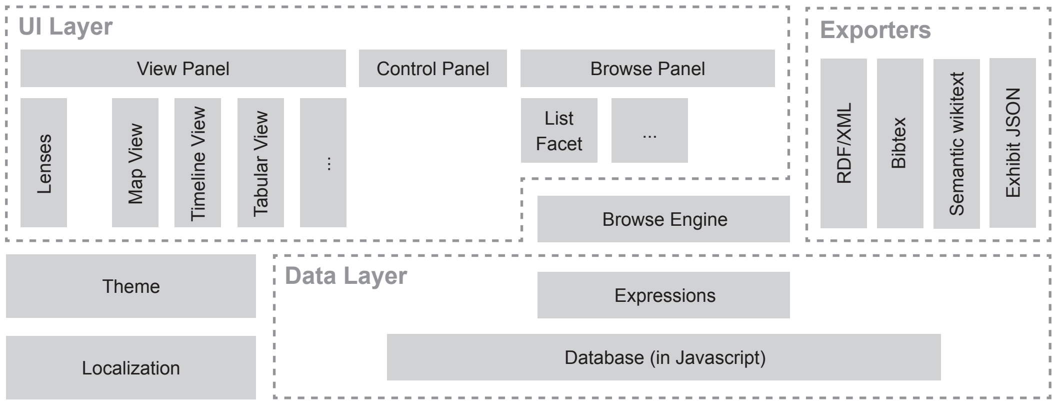


Figure 10. Exhibit's architecture.

Exhibit's architecture is illustrated in Figure 10. At the bottom is a database implemented in Javascript. The database and the expression facilities form the data layer on which much of the rest of the system depends.

There are several points of extensibility. More views can be added. More exporters can be registered. The Browse Panel can also be extended with facets that don't just list their choices but specialize for the properties that they are configured with, such as showing a calendar for a date property.

The localization component encapsulates localized UI resources, including text strings, images, styles, and even layouts. This is only our early attempt—internationalizing a framework that generates user interfaces at run-time is very difficult. We note that even HTML is biased for English. For example, bold and italics, which have native syntax in HTML, are foreign concepts to most Asian scripts.

Exhibit's database is the main bottleneck for Exhibit's performance scalability at the moment. Each exhibit handles up to a few hundred items and still remains usable. However, we expect that our own implementation can be optimized further, that browsers' Javascript engines will get faster, and that computer hardware will also speed up in time. Furthermore, web browsers might expose local database storage APIs to web pages in the future, which can replace our Javascript-based implementation.

9. DISCUSSION

The sale distribution of books on Amazon is a long-tail distribution. When books are ranked by popularity, plotting their sale records yields a downward sloping, concave curve that approaches the x axis but does not fall to zero for a long, long time . There are hundreds of thousands of unpopular books, each selling only a few to a few hundred copies, whose combined profit rivals that of the few hundred books that sell millions of copies. This observation is explored in Anderson's book, "The Long Tail." [10]

Not only do the books themselves follow a long-tail distribution, but so do the topics that those books cover. If we rank the various kinds of information that people want to publish on the Web by their popularity, we would also get a long tail (Figure 11): a few dozens of readily recalled kinds of information such as consumer products, news articles, events, locations, photos, videos, music, and software projects populate the massive head of the curve while hundreds of thousands of unpopular topics, such as sugar packet collection and lock picking, spread thinly over the long, long tail.

Clustering search engines such as clusty.com show dozens of sites of each unpopular topic, like sugar packet collection and lock picking. Wikipedia's several thousands of categories and DMoz's half a million categories show how long the tail is.

Here in the tail, we find a large number of potential homesteaders for our Semantic Web. We hope that Exhibit appeals to these authors and thus achieve widespread adoption. We now discuss the implications of this optimistic outcome.

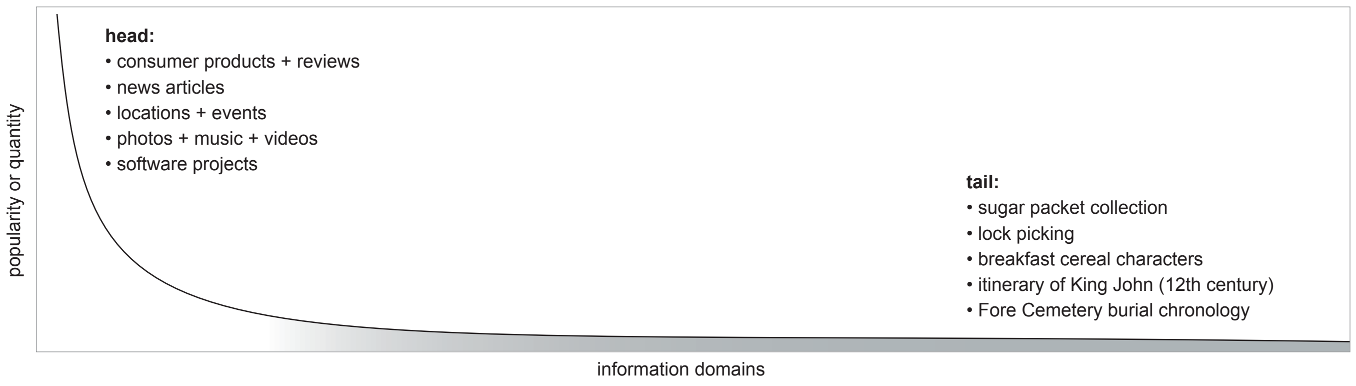


Figure 11. The Long Tail of Information Domains.

From a Semantic Web point of view, millions of exhibits, with different ontologies, do not immediately give rise to a globally coherent "web of data." However, these exhibits will have achieved two improvements over the current state of the Web. First, structured data previously encoded in HTML but now in exhibits is much, much easier to harvest. There is no need to develop sophisticated scrapers for each of the millions of hand-built web sites—likely an impossible task at that scale. Instead, by baiting them with good UIs, we convince authors to structure the data for us. This improvement can be seen as the first step to structurizing the Web, done by humans to help out the machine processing that follows. Getting the data into machine-readable structured representation means that we can focus on the semantic problem of ontology alignment instead of the blurry syntactic problem of scraping. Second, by including Exhibit, the current web browser is automatically "upgraded" to be a structured data-aware browser at no cost to the user nor to the author. This improvement can be seen as the first step in morphing the web browser into the semantic web browser without requiring users' buy-in.

Even without solving ontology alignment, the data harvested from these exhibits can already improve web searching and browsing. For example, one can now search for data having a particular Exhibit type, e.g., "Character", and a particular property, e.g., "brand". The search engine might confound breakfast cereal characters with action hero characters, but not with movie characters who have no "brand." These extra structured data-based query terms will help reduce the search space.

Indeed, moving all information on the Web to the Semantic Web is a grand challenge. The insight into the long tail of information domains has inspired both this work and our previous efforts, Piggy Bank [14] and Sifter [15]. Piggy Bank and Sifter target *users* at the *head* of the distribution where the information comes from large publishers and is relatively well formatted by server-side technologies. By scraping that information and using it to provide more values to users, we hope to "trick" users into converting information at the head into Semantic Web format. At the tail, the existing information is too unstructured to scrape reliably. So at the *tail*, Exhibit enrolls *authors*, instead of users, into converting that information into more structured form. Thus, we have the whole distribution curve covered.

10. CONCLUSION

In this paper, we presented Exhibit, a lightweight structured data publishing framework that targets authors at the tail of the information domain distribution on the Web. These authors are motivated to publish their data although they have few skills and fewer resources. However, if given the right tools that yield instant gratification for little efforts, these authors can help structure-ize information at the tail of the Web—information that has previously been authored in HTML by hand and remains resistant to automatic scraping.

If Exhibit achieves wide adoption, we will study the structured data medium that it creates. In particular, we will explore ways for users to *mash up* data from several exhibits and perform ontology alignment on the fly to achieve value from such aggregation.

11. ACKNOWLEDGMENTS

We are grateful to the members of the Haystack group, the User Interface Design group, the Simile team, and the W3C at MIT CSAIL for their insightful comments on our work. This work has been funded by [funders].

REFERENCES

[1] Babel. http://simile.mit.edu/babel/.
[2] DabbleDB. http://dabbledb.com/.
[3] Google Base. http://base.google.com/.
[4] Google Maps. http://maps.google.com/.
[5] Greasemonkey. http://greasemonkey.mozdev.org/.
[6] Introducing JSON. http://www.json.org/.
[7] Simile Timeline. http://simile.mit.edu/timeline/.
[8] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
[9] Topher's Breakfast Cereal Character Guide. http://www.lavasurfer.com/cereal-guide.html.
[10] Anderson, C. The Long Tail: Why the Future of Business is Selling Less of More. New York: Hyperion, 2006.
[11] Bizer, C., E. Pietriga, D. Karger, R. Lee. Fresnel: A Browser-Independent Presentation Vocabulary for RDF. ISWC 2006.
[12] Bolin, M., M. Webber, P. Rha, T. Wilson, and R. Miller. Automation and Customization of Rendered Web Pages. UIST 2005.
[13] Hildebrand, M., J. van Ossenbruggen, L. Hardman. /facet: A Browser for Heterogeneous Semantic Web Repositories. ISWC 2006.
[14] Huynh, D., S. Mazzocchi, and D. Karger. Piggy Bank: Experience the Semantic Web Inside Your Web Browser. ISWC 2005.
[15] Huynh, D., R. Miller, and D. Karger. Enabling Web Browser to Augment Web Sites' Filtering and Sorting Functionality. UIST 2006.
[16] Oren, E., R. Delbru, S. Decker. Extending faceted navigation for RDF data. ISWC 2006.
[17] schraefel, m. c., Karam, M. and Zhao, S. mSpace: interaction design for user-determined, adaptable domain exploration in hypermedia. AH 2003: Workshop on Adaptive Hypermedia and Adaptive Web Based Systems.
[18] Sinha, V. and D. Karger. Magnet: Supporting Navigation in Semistructured Data Environments. SIGMOD 2005.
[19] Völkel, M., M. Krötzsch, D. Vrandecic, H. Haller, R. Studer. Semantic Wikipedia. WWW 2006.
[20] Yee, P., K. Swearingen, K. Li, and M. Hearst. Faceted Metadata for Image Search and Browsing. CHI 2003.